DOCUMENT RESUME

ED 338 217 IR 015 242

AUTHOR Taylor, Karen A.
TITLE An Annotated Bibliography of Current Literature Dealing with the Effective Teaching of Computer Programming in High Schools.
PUB DATE Jun 91
NOTE 54p.; Student research project, Indiana University at South Bend.
PUB TYPE Information Analyses (070) -- Reference Materials - Bibliographies (131) -- Dissertations/Theses - Undetermined (040)

EDRS PRICE MF01/PC03 Plus Postage.
DESCRIPTORS Algorithms; Annotated Bibliographies; Authoring Aids (Programing); Cognitive Processes; *Debugging (Computers); Flow Charts; High Schools; *Problem Solving; *Programing; *Teaching Methods

ABSTRACT

This review of the literature and annotated bibliography summarizes the available research relating to teaching programming to high school students. It is noted that, while the process of programming a computer could be broken down into five steps--problem definition, algorithm design, code writing, debugging, and documentation--current research was concentrated on three of the steps only, i.e.: (1) algorithm design (12 references); (2) code writing (7 references); and (3) debugging (6 references). The first of this paper's four major sections provides a statement of the problem, the purpose of the study, its limitations, and definitions of terms used. The second section presents a review of the literature in the form of an annotated bibliography, and the third summarizes and discusses the findings of the literature review. An alphabetical list of 32 cited works concludes the paper. (DB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

This document has been reproduced as received from the person or organization originating it.

Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED338217

An Annotated Bibliography of Current Literature Dealing with the Effective Teaching of Computer Programming in High Schools

by

Karen A. Taylor

Research Project in Secondary Education

Charles R. DuVall, Ph.D.

June 8, 1991

IR015242

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Karen A. Taylor

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

BEST COPY AVAILABLE

TABLE OF CONTENTS

I. Introduction

Statement of Problem.....1

Purpose of Study.....1

Limitations.....2

Definition of Terms.....3

II. Review of Related Literature

Algorithm Design.....6

Code Writing.....15

Debugging.....19

III. Summary, Conclusions and Recommendations

Summary.....22

Problem Specification.....22

Algorithm Design.....23

Code Writing.....30

Debugging.....35

Documentation.....42

Conclusions.....43

Recommendations.....46

IV. Works Cited.....49

V. Student Qualifications.....52

Note: Part V was not included in the copy received by ERIC.

STATEMENT OF PROBLEM

The complex task of programming a computer could be broken down into five distinct steps, each of which required different skills and strategies from the student and different instructional methods from the teacher.

PURPOSE OF THE STUDY

Computer programming could be considered a fairly new educational discipline. Current research seemed to be divided into two fairly distinct areas. Researchers either evaluated programming as a means of promoting the cognitive skills of problem solving, or determined the most effective methods for teaching programming.

Of immediate interest to teachers, was the research dealing with instructional methods of teaching programming. Current research had just started to define the concepts and procedures necessary for success in some areas of programming. The high school teacher, however, would find that, as Linn stated, "no clear guidance on how to teach programming exists" (1987, 468). This paper looked at the research available that related directly with teaching programming.

LIMITATIONS

This paper looked at research on topics specifically useful to a high school instructor teaching computer programming classes. It made no attempt to justify neither the existence nor the value of such classes. Nor did it pursue the question of which programming language should be taught in high school.

Problem solving was considered an integral part of the program development process, particularly the algorithm design and debugging phases. The emphasis here was on problem solving approaches to programming rather than the transfer of those skills to other domains.

DEFINITION OF TERMS

algorithm	a specific procedure that, when followed, yields a successful solution to a problem. (Ingram, 215)
bug	an error in a computer program. (Good)
code	in programming instruction, to write the instructions constituting a program. (Good)
compile	to produce a machine language routine from a routine written in source language (Good)
computer programming	the setting up of a computer to solve a problem; the breakdown of a problem of any complexity into a program which consists of a series of instructions usually expressed in a compiler code. (Good)
debug	to locate and correct any errors in a computer program, to detect and correct malfunctions in the computer itself, or to test-run and check out a program of machine instructions for a digital computer for the purpose of eliminating mistakes. (Good)
documentation	in computing, a term for all the various manuals, program specifications and other documents that the operators of a computer need to enable them to use the machine properly. (Ellington)
feedback	knowledge obtained from tryout of programmed materials that may be used by the programmer in making subsequent revisions. (Good)
flow of control	the order in which the computer executes statements in a program. (Linn, 1985, 15)

flow chart	a diagrammatic representation of a sequence of operations required to carry out some procedure; steps in the procedure are usually in rectangles, circles, etc., with arrows leading from one figure to the next; of particular importance in making a flow chart is the insertion of instructions for making decisions and for repeating part of the procedure. (Good)
heuristic	a general strategy for approaching a problem, and as such cannot be applied with the precision of an algorithm. (Ingram, 215)
instructional strategies	are general design plans that mainly differ in their control of students' processing-load but that all pursue the same high-level goals which teachers consider desirable. (Van Merrienboer, 253)
instructional tactics	are specific design plans that prescribe methods to reach desired learning outcomes under given circumstances. (Van Merrienboer, 253)
iterative	describing a procedure or process which repeatedly executes a series of operations until some condition is satisfied. (Good)
on-line	directly connected to and operating under the control of the automatic computer. (Good)
procedural reasoning	includes understanding conditional relationships, temporal sequencing, hypothetical deduction and planning. (Kurland, 1984, 8)
pseudocode	a series of commands or instructions, used by programmers in writing a program, which have the form of regular commands or instructions but the function of defining operands, specifying origins or otherwise directing the translation process. (Good)

semantic knowledge	1) refers to the meaning of a statement or of a program or program module. (Dyck, 16); or 2) hierarchically organized knowledge with concepts ranging from lower levels, such as the assignment statement, to higher levels, such as the pattern of code for finding the mean of an array. (Van Merrienboer, 256)
simulation	an initiative type of data processing in which a computer is used to implement an information model of some entity such as a chemical process; information enters the computer to represent the factors entering or affecting the real process and the computer then produces information that represents the result of the process, with the processing done by the computer representative of the process itself. (Good)
structured programming	a systematic way of designing, building, validating and documenting computer programs which, if carried out correctly, leads to the production of error-free, efficient and reliable software. (Ellington)
syntactic knowledge	1) a user's knowledge of lexical units and knowledge of rules for combining lexical units. (Dyck, 16); or 2) unorganized knowledge of low level details, such as the syntax of language features and the names and arguments of functions. (Van Merrienboer, 256)
template	a sequence of code, involving more than one language feature, that can be used to perform a complex function. (Linn, 1985, 15)
variable	a quantity which varies, or is capable of varying or being varied. (Ellington)

REVIEW OF RELATED LITERATURE

Structured programming could be divided into five steps: 1) problem definition; 2) algorithm design; 3) code writing; 4) debugging; and, 5) documentation (Kurland, 1984, 2; Goktepe, 168; Webb, 1985, 186; Dalbey, 1985, 254; Moursund, 4). Current research was concentrated on the second, third and fourth phases only. No research was found pertaining to the problem definition or the documentation steps. Reviews of the algorithm design, code writing and debugging phases follow.

ALGORITHM DESIGN

Kurland, D. Midian, Clement, Catherine, Mawby, Ronald and Pea, Roy D. "Mapping the Cognitive Demands of Learning to Program," A Symposium: Annual Meeting of the American Educational Research Association, New Orleans, Louisiana, Technical Report No. 29, Apr. 1984, 1-35.

To determine the cognitive skills necessary for programming, the authors first investigated all the factors that defined the programming process. The age of the programmer, expected level of expertise, characteristics of a specific language, quality of the programming environment and style of instruction all effected the cognitive demands required.

Studying high school students learning Logo in a self discovery atmosphere, some relationship was found between programming proficiency and the skills of procedural reasoning and decentering. Procedural reasoning was thought to be necessary for designing and following the flow of control of a program. Decentration was thought to be helpful in debugging, when differentiating between what the program should do and what the computer actually did.

The authors found that even though students did well on the program comprehension test, they did not produce well-planned, structured code on the program generation test. It was concluded that while programming could invoke high level thinking skills, today's interactive programming environment did not foster their development.

Kurland, D. Midian, Pea, Roy D., Clement, Catherine and Mawby, Ronald. "A Study of the Development of Programming Ability and Thinking Skills in High School Students," Journal of Educational Computing Research, 1986, 2 (4), 429-458.

This study examined pre-college students, focusing on 1) predictors of programming ability, 2) programming proficiency after two years of study, and 3) transfer of programming skills to mathematics problem-solving.

The study did verify that procedural reasoning and decentering skills, along with the ability to evaluate values of variables, the ability to translate word problems into symbolic equations and the ability to design and comprehend an algorithm, all promoted the development of programming skills.

Extensive pre- and post-testing seemed to indicate, however, that after two years the students had not attained a very high level of programming expertise. They could do simple, short, linear programs, but had difficulty with iterative, conditional and recursive structures. Adequate debugging skills were also lacking.

Observations of students as they worked indicated that they frequently shared ideas and code, used a trial-and-error approach to design and immediately asked for help when stuck.

There appeared to be little transfer of programming skills to other domains that required similar cognitive abilities.

Van Merrienboer, Jeroen J. G. and Krammer, Hein P. M.
"Instructional Strategies and Tactics for the Design of Introductory Computer Programming Courses in High School," Instructional Science, 1987, 16 (3) 251-85.

Upon investigation of textbooks and curricula, this study determined that the three most commonly used instructional strategies for the teaching of introductory computer programming were the Expert, the Spiral and the Reading approaches. The authors then evaluated the extent to which each of these approaches were compatible with six instructional tactics which had been proven effective in teaching programming. The three declarative tactics included explicitly presenting concrete computer models, programming plans and design diagrams. Three procedural tactics included providing annotated, worked-out examples of code, extensive practice in basic skills and task variation of both problem types and programming skills.

The Expert approach emphasized top-down algorithm and program design, with students studying large complex programs from the start of instruction. This approach was the least successful in controlling the students' processing loads. It was highly compatible with the tactics of providing program plans and design diagrams, but at the cost of the coding aspect, program comprehension and basic skills. Since program generation was the primary student activity, task variation was low.

In the Spiral approach, students were simultaneously presented with syntactic and semantic knowledge in small incremental steps. As the students mastered basic skills, program requirements progressed from trivial to more complex, with design skills not required until late in the course. Instructional tactics highly compatible with this approach were the use of computer models, program plans and skills practice. Again, program generation was the primary goal, with little task variation.

The Reading approach was a four step program. Students were to first, run programs, observing their behavior and evaluating strengths and weaknesses; second, hand-trace programs and predict output; third, modify and amplify existing programs; and finally, generate their own programs. With the exception of design diagrams, this approach was highly compatible with all the instructional tactics.

Dalbey, John and Linn Marcia C. "Cognitive Consequences of Programming: Augmentations to BASIC Instruction," Journal of Educational Computing Research, 1986, 2 (1), 75-93.

As part of the Assessing the Cognitive Consequences of Computer Environments for Learning (ACCCEL) Project, Dalbey and Linn added two units to the introductory BASIC classes at two middle schools. One augmentation, Spider World, was a programming environment designed to emphasize the cognitive and motivational factors of programming. With simplified syntax and structured flow of control commands, students progressed quickly to problems that demanded the use of design skills. Type Attack was a typing tutor arcade program, requiring problem solving strategies to win the game. The augmentations were taught first, then the students went on to BASIC.

In post-tests, students in the Spider World group by far surpassed the Type Attack group and outscored the control groups in BASIC syntax knowledge. Both augmented groups seemed more adept at using a programming-like environment and were more enthusiastic about programming.

In general, student scores were much lower and varied more widely than had been expected, or than could have been accounted for by the augmentation. The authors concluded that while Spider World produced some success, other undetermined factors appeared to have a larger effect on programming outcomes.

Ingram, Albert L. "Instructional Design for Heuristic-Based Problem Solving," Educational Communication and Technology Journal, Win. 1988, 36 (4), 211-30.

Ingram contended that problem solving was an umbrella term that applied to many different strategies and skills, each of which could be used alone or in combination in a great variety of situations. There was also very little research available on procedures to design and test educational modules for teaching specific problem solving skills.

He designed a procedure for teaching the heuristics involved in the computer programming problem solving technique of top-down design. A heuristic was defined as "a general strategy for approaching a problem", in contrast to an algorithm, which was a "specific procedure that, when followed, [yielded] a successful solution." The heuristics involved in programming included understanding the structure of the problem and the solution, and generating and testing alternative solutions.

Working with graduate students in a BASIC programming course, Ingram reported some success with the explicit teaching of top-down design skills that resulted in more efficient code written in considerably less time.

Dalbey, John, Tourniaire, Francoise and Linn, Marcia C. "Making Programming Instruction Cognitively Demanding: An Intervention Study," Journal of Research in Science Teaching, May 1986, 23 (5), 427-36.

As part of the ACCCEL (Assessing the Cognitive Consequences of Computer Environments for Learning) project, the authors intervened during the second semester of a junior high class in BASIC programming. The objective was to give the students instruction on using structure diagrams to plan programs, then to return at a later date to see if the students' problem-solving techniques had changed. The teacher up to this point had not taught, modeled or emphasized program design procedures.

After one week, the students were able to use structured diagrams with simple standard problems, but had difficulty with more complex problems that deviated from given examples. They also could not refine a general problem situation down to detailed machine acceptable steps. The major difficulty was that the students failed to see the design process as beneficial to their programming. They preferred a trial and error method at the terminals.

The study recommended that teachers emphasize planning from the beginning, perhaps by stressing comprehension of medium length programs and their design, rather than language syntax.

Scanlan, David and Clark, Leisa. "An Empirical Investigation of Flowchart Preference," Journal of Computers in Mathematics and Science Teaching, Win. 1988-89, 8 (2), 56-64.

In previous studies, the authors determined that between 70 and 80 percent of students preferred structured flowcharts over pseudocode as an aid to the understanding of short, complex algorithms. In this study he found that the students' actual behavior did indeed match their stated preferences.

The students' claims that flowcharts were easier to understand were consistent with the hypothesis that the graphic-algorithmic properties of flowcharts tapped the neuro-cognitive resources of both brain hemispheres, while the highly verbal pseudocode mostly taped the resources of the left hemisphere."

The study also found that the older the student, and the higher the Grade Point Average of the student, the more likely the student would prefer structured flowcharts. Also more girls than boys preferred flowcharts.

The authors concluded that more research was needed in assessing the value of graphic and verbal presentations and in the interim, perhaps textbook publishers and teachers should take students' preference into consideration and use more flowcharts.

Ross, Steven M. and McCormick, Deborah, "Computer Access and Flowcharting as Variables in Learning Computer Programming," Report: Annual Meeting of the Association for Educational Communications and Technology, Feb. 1989, 1-10.

High school students studying BASIC programming as part of a one semester Computer Literacy course were divided into four groups, each having either unlimited or restricted computer access time along with either mandatory or optional flowcharting assignments. As a side issue, the effect of gender and ability on each of these learning atmospheres were studied.

The results surprised the authors. The low ability groups did better with unlimited computer access; however, the medium and high ability students actually performed better with restricted access.

Students scored better on post-tests when not required to submit flowcharts. They did not have time to master the techniques required for flowcharting and appeared to regard it as a separate task instead of a planning tool.

In general, males and females scored about the same on all tests, except girls did slightly better when computer access time was restricted.

MacGregor, S. Kim. "The Structured Walk-Through," Computing Teacher, Jun. 1988, 15 (9), 7-10.

As a method for teaching structured programming principles, emphasizing planning and encouraging student collaboration, MacGregor suggested implementing a version of the structured walk-through that was used in the professional setting. Students were to analyze problems and write pseudocode individually, critique the plans in small groups, code and debug the programs separately and finally evaluate each other's finished efforts. The students were given clearly defined goals for each step in the process. They were evaluated by the instructor for evidence of modularity, adequate documentation, utilization of program concepts emphasized in class and correctness and quality of the output.

MacGregor used the structured walk-through in one of two high school Pascal classes and found students using this method were better able to design algorithms using pseudocode, wrote more modular programs and exhibited a more positive attitude toward programming. These students also spent more time planning their programs, less time debugging, and interacted more with peers than with the teacher. Both classes demonstrated equal ability to predict the output from Pascal code.

Webb, Noreen M. and others. "Problem Solving Strategies and Group Processes in Small Groups Learning Computer Programming," American Educational Research Journal, Sum. 1986, 23 (2) 243-61.

In a study of middle school students learning BASIC, the authors confirmed that group learning of programming seemed to have positive results.

Students that actively participated in group discussions about planning and debugging strategies demonstrated successful programming outcomes. There was no correlation between group interaction and knowledge of syntax but a significant correlation between group work and student's behavior, whether the planning and debugging activities were carried out on a specific level or a more abstract level. Group interaction then appeared to have an indirect influence on a student's programming achievement.

An interesting result of the study showed that student group work was positively related to programming achievement if students discussed plans with each other, but negatively or not related to success if they verbalized their strategies to an instructor.

As a result of this, Webb and Lewis suggested that instructors structure their courses around group interaction, intervening as little as possible so that students would be forced to discuss problem solving strategies with each other.

Linn, Marcia C. "Autonomous Classroom Computer Environments for Learning. Progress Report and Annotated Bibliography," May 1988, 1-13.

Linn reported on two studies done as part of the Autonomous Classroom Computer Environments for Learning (ACCEL) project. The first involved communicating expert design techniques to student programmers, and the second reported on the effect of classroom characteristics on student learning.

The behavior of experts was analyzed and found to reflect three techniques: 1) experts had a repetoire of templates; 2) through experience, they combined templates into a library of procedures; and 3) they increase their knowledge of programming by reading the code of other programmers. Expert solution sets for several complex problems were made that included 1) lists of templates, 2) an illustration of the process to select the best template, 3) an illustration of how to search for bugs, 4) instructions for piece by piece implementation of the program, and 5) testing procedures. Groups of high school Pascal students were engaged in a variety of activities, including writing their own code for the problem, reading the expert solution, executing the expert solution and answering study questions about the solution set. Linn found that students learned more about programming if they participated in all four activities.

The classroom characteristics that correlated with student learning included computer access, explicit teaching and immediate feedback. Students that had more access time on the computer were more successful. Individual or small group instruction proved more effective than large group lectures. Students benefited most when assignments were returned promptly with comments.

Linn, Marcia C. "The Cognitive Consequences of Programming Instruction in Classrooms," Educational Researcher, May 1985, 14 (5), 14-16, 25-29.

As a result of research, interviews and observations, Linn identified a chain of cognitive accomplishments that could be learned from programming courses. The results of the ACCCEL (Assessing the Cognitive Consequences of Computer Environments for Learning) Project were then analyzed to discern how far along that chain students who studied programming had progressed. The cognitive steps included 1) learning the language features, 2) studying design techniques, including templates and procedural skills, and 3) generalizing problem-solving skills.

Middle school classes in BASIC for 2400 students were categorized as typical or exemplary, those where students were explicitly taught how to design programs. The exemplary classrooms generally had more experienced teachers and higher than average ability students. Teachers in the exemplary classes explicitly taught design templates and procedural skills such as debugging and reformulation techniques.

After twelve weeks of study, there was a tremendous difference in student performance, ranging from 17% to 86% on a final programming test. Students in typical classes were still at the language feature link on the chain of development. Students in the exemplary classes had mastered the language features and could design programs requiring 10 to 50 lines of code. General ability, in-school access to computers, and home computers had an effect on the performance of the typical classroom students, but not on the exemplary class students. Previous computer experience, attitude toward computers and gender did not seem to have an effect on any of the classes.

A group of talented programmers, the top one percent of the students, was singled out based on their programming success. While most of this group were high ability students, some middle and low level ability students from the exemplary classes were included. It was noted that while females comprised only 37% of the entire student sample, they made up 60% of the top talented group.

Linn concluded that one programming course did not move students very far along the chain of cognitive accomplishments. Explicit teaching of design and procedural skills was effective. More experienced teachers tended to employ these more effective teaching methods.

CODE WRITING

Magliaro, Susan and Burton, John K. "Adolescents' Chunking of Computer Programs," Paper presented at the Annual Meeting of the American Educational Research Association, Apr. 1986, 1-19.

This study was designed to investigate how children organized their knowledge of the programming process in memory. Students at a computer camp were divided into beginner, intermediate and advanced groups and were correspondingly taught BASIC, advanced BASIC and Pascal. All instruction emphasized structured programming skills. As a test, students were given chunks of BASIC programs, some scrambled, to study, then recall.

Beginners were able to correctly recall some statements, a few program chunks, and no scrambled programs. They seemed to rely on rote recall based on their syntactic knowledge or on the physical features of the words. The intermediate students excelled at recalling program chunks, while the advanced group were best at recalling scrambled programs. Both the intermediate and advanced groups seemed to draw on their experience, attempted to make sense out of the chunks, and memorized procedural features. The advanced students were confident enough in their programming skills to point out that the scrambled code was incorrect.

Dyck, Jennifer L. and Mayer, Richard E. "Teaching for Transfer of Computer Program Comprehension Skill," Journal of Educational Psychology, Mar. 1989, 81 (1), 16-24.

The authors examined two theories of learning programming languages. The syntax-dependent theory suggested that the syntax and semantics of a programming language were interdependent and needed to be learned simultaneously, the semantic concepts within the syntax constraints. The syntax-independent theory contended that semantics may be learned first, independent of syntax.

Two groups of college students, with no computer programming experience, were tested. A control group was taught BASIC syntax and semantics simultaneously. A transfer group was first taught semantics in English, then BASIC syntax. The transfer group learned BASIC just as fast as, and in fact, progressed quicker than, the control group. In program generation, the transfer group excelled in producing semantically correct code, while the control group proved better at generating syntactically correct programs.

The study suggested that students could benefit from first learning the underlying concepts of a programming language in their own natural language, freed from the constraints of the language's syntax.

Kagan, Dona K. "Learning How to Program or Use Computers: A Review of Six Applied Studies," Educational Technology, Mar. 1988, 28 (3), 49-51.

In this review of six studies that were conducted over a four year period involving 1,009 undergraduate students, Kagan summarized the evidence that was found linking natural language skills with success in learning a programming language. There was a significant difference between the cognitive tasks of learning the basic language syntax, comprehending the semantics of a FORTRAN program and composing original code. These tasks, Kagan hypothesised, were analogous to the skills associated with natural language proficiency; a knowledge of vocabulary, the ability to comprehend written material and the ability to write cohesive text.

Kagan found a significant relationship between achievement in FORTRAN programming and a tendency to organize linguistic information analytically. Females tended to draw more on their linguistic abilities than did males.

With these results, Kagan suggested that programming instructors treat understanding and writing code as two different skills, rather than attempt to teach writing through the use of model programs, written by someone else.

Linn, Marcia C., Sloane, Kathryn D. and Clancy, Michael J., "Ideal and Actual Outcomes from Precollege Pascal Instruction," Journal of Research in Science Teaching, 1987, 24 (5), 467-490.

This was the first in a series of studies by the authors on the relationship between instructional practices and learning outcomes in computer programming.

Fourteen full year, advanced placement high school Pascal classes, taught by 12 different teachers were used. During the year the students' backgrounds and general abilities, along with the teachers' instructional methods were assessed. At the end of the year, classes were divided into three groups according to an evaluation of the students' programming proficiencies. The authors looked for similarities and differences in the various instructional methods to explain the differences in learning outcomes.

The study suggested that more on-line machine access, explicitness of instruction and detailed feedback were the three factors that contributed to student success in learning programming. A positive relationship was found between programming proficiency and computer access, individual or small group work with the teacher, quick return of assignments and programs from the teacher, and requirements of written plans before coding.

Pintrich, Paul R. and others. "Students' Programming Behavior in a Pascal Course," Journal of Research in Science Teaching, May 1987, 24 (5) 451-66.

Pintrich studied experienced high school students in an Advanced Placement Pascal programming course. Using a list of nineteen observable behaviors, Pintrich kept track of both the frequency and duration of each behavior for a minimum of two hours for each student, scattered throughout the semester in ten minute increments.

He found that students did not engage in planning and spent very little time writing original code. The majority of their time was spent editing code or running and listing their programs. The predominent debugging technique was a trial and error method; the students would make one change or correct one error, then compile and run their program. Students did stay on task the majority of the time, with little talking or time away from the machine. Students spent a fair amount of time getting help from their text.

The behaviors reflected the instructional atmosphere of the classroom. The instructors' main goal was to teach Pascal language features. They did not provide much direct instruction in design or procedural skills. Debugging skills were not taught and group interaction was not allowed.

Webb, Noreen M. "Cognitive Requirements of Learning Computer Programming in Group and Individual Settings," AEDS Journal, Spr. 1985, 18 (3), 183-94.

This study of junior high school students learning BASIC reported no appreciable difference in programming outcomes between pairs of students working together and students working individually. Webb tested knowledge of the function of commands, knowledge of syntax, ability to interpret programs and ability to generate programs.

The study did, however, show that different cognitive characteristics effected student's success in either the group or individual setting. All students with good mathematics and verbal inference ability were highly successful in learning syntax rules, but did better generating code when working alone rather than in groups. Nonverbal reasoning ability was a good predictor of knowledge of syntax in a group setting. Students with good spatial ability did well interpreting programs when working alone, but were more successful writing programs working in a group. Older students interpreted programs better working in a group.

Webb did not find any predictor of knowledge of the function of commands and no one characteristic predicted success in all areas of programming.

Cook, Curtis, Bregar, William and Foote, David. "A Preliminary Investigation of the Use of the 'Cloze' Procedure as a Measure of Program Understanding," Information Processing and Management, 1984, 20 (1-2), 199-208.

This study of college computer science majors investigated measures of program understanding. The authors stated that the comprehensive quiz was probably the most commonly used and accepted method of testing. It took considerable time, however, to construct an objective test with appropriate items to test both low-level and high-level comprehension.

In the "cloze" procedure, the students were given a program with some of the tokens replaced with blanks, and were required to fill in the missing reserved words, operators, operands or parentheses. The test was trivial to construct, was not limited by the size of the program and could be scored objectively.

The authors found the scores on the "cloze" tests were a close approximation of the scores obtained with a comprehensive test for all three levels of programming ability studied. The "cloze" test was simpler and quicker to construct, provided an objective standard for comparison and could be applied to small or large programs.

DEBUGGING

Klahr, David and Carver, Sharon McCoy. "Cognitive Objectives in a LOGO Debugging Curriculum: Instruction, Learning, and Transfer," Cognitive Psychology, Jul. 1988, 20 (3), 362-404.

Using a computer simulation to model a LOGO debugger, the authors defined five steps in the debugging process, which they then directly taught to a group of elementary school students. The students learned to successfully apply the debugging strategies and continued to decrease their debugging time even after a four month period with no further programming instruction.

The plan required four sources of information: a) a specification of the desired outcome, b) a listing of the program, c) the actual output produced by the program, and d) the student's cache of programming knowledge. The debugging process included five phases: 1) The program plan and the actual outcome was matched to determine what, if any, discrepancies existed. 2) The discrepancy was described in enough detail to identify the specific type of bug that might have caused the error. 3) A knowledge of the program's structure was needed to determine which portion of the program was most likely to contain the bug. 4) The suspected code was examined in order to locate the bug. 5) The bug was corrected and the program was run and re-evaluated.

Newby, Michael. "Teaching Introductory Programming Using a Language-intelligent Programming Environment," In Barrett, John, and Hedberg, John (Eds.) "Using Computers Intelligently in Tertiary Education," A Collection of Papers Presented to the Australian Society for Computers in Learning, Sydney, New South Wales, Australia, Nov.-Dec. 1987, 52-57.

This paper described a programming environment, ALICE, from Looking Glass Software, for Pascal programmers. The editor was designed to save entry, coding and debugging time.

When a key word was entered, a template was displayed on the screen. The programmer followed the template, replacing the place-holders with appropriate variables or commands. The system was language-intelligent and checked for syntax errors and type compatibility as text was entered. The editor flagged undefined variables and could do some semantic error checking. One-step execution was a debugger option, displaying both the code and the corresponding variable values on the screen simultaneously.

Novice programmers found the system easy to use and were able to concentrate on problem solving techniques rather than syntax rules. Experienced programmers found the system restrictive.

Samurcay, Renan. "Learning Programming: An Analysis of Looping Strategies Used by Beginning Students," For the Learning of Mathematics-An International Journal of Mathematics Education, Feb. 1985, 5 (1), 37-43.

Samurcay contended that programming activity covered a large hierarchy of concepts and procedures of varying degrees of complexity. In this article, he examined the relationship between the concepts of variable, assignment and looping.

By studying students as they tried to solve a series of programming problems, Samurcay concluded that a student's mathematical conceptions were insufficient for building programming concepts. Students did not fully understand a variable's role as a dynamic, changing value in a program as opposed to the unique, static value it had in mathematics. Assignment statements in programming were asymmetric expressions rather than equalities.

The program loop concept, according to Samurcay, used the notions of variable and assignment in their "full programming sense," and the problems students had in designing the transformation procedures in a loop stemmed from their misconceptions of these basic programming concepts.

Pea, Roy D. "Language Independent Conceptual 'Bugs' in Novice Programming," Journal of Educational Computing Research, 1986, 2 (1), 25-36.

In this article, Pea examined the harmful effects of the transfer between natural language and programming languages, and how that transfer lead to misconceptions among novice programmers. When students did not fully understand exactly how a computer interpreted their instructions, they reverted back to their natural language as their frame of reference. In doing so, they tended to attribute human conversational characteristics to the computer.

There were three main classes of "bugs" or misunderstandings. The "parallelism bug," assumed that several lines of code in a program could be active at the same time, just like a human could process several lines of a conversation at one time. The "intentionality bug" assumed that the computer could look ahead, or anticipate, what would happen in the program. The "egocentrism bug" assumed the computer could fill in or supply parts of code that the student had left out, similar to the way a human could interpret what was meant from what was said.

Pea stated that much more research was needed on how to help students see that "computers read programs through a strictly mechanistic and interpretive process," not a human one.

Pea, Roy D. and others. "The Buggy Path to the Development of Programming Expertise," Focus on Learning Problems in Mathematics, Win. 1987, 9 (1), 5-30.

Pea continued his research, reported in an earlier study, in the classification and description of "bugs" in novices' programs. Some language-independent conceptual bugs arose from a lack of metacognitive skills. Students spent little time evaluating their problem-solving processes, which lead to program errors, poorly planned programs, inadequate testing and lack of documentation. A second area of conceptual bugs involved program design; understanding the goals behind each language construct and knowing when to use a specific construct rather than how to use it. Students also had difficulty with merging different constructs together into a single unified plan.

Two types of language-dependent conceptual bugs included a lack of knowledge of the language syntax, and the inability to apply knowledge the students already had to a slightly different situation.

Pea concluded that teachers needed to adapt research methods to discover problem areas, and then the potentially buggy areas needed to be explicitly confronted by both the students and the instructors.

Sleeman, D. and Gong, Brian. "From Clinical Interviews to Policy Recommendations: A Case Study in High School Computer Programming. Study of Stanford and the Schools Technology Panel," 1985, 1-7.

This series of four studies focused on novice programming skills from a cognitive science perspective. Through clinical interviews with high school BASIC and Pascal students, the first study identified and classified over 30 misconceptions that students had about the nature of the computer's processing operation and about the syntax and semantics of the programming language.

A second study reported successful methods of remediating these misconceptions. They included explicit training in the syntax of the language, requiring students to learn to read programs before attempting to write them and providing interactive feedback from the computer.

Sleeman contended that some misunderstandings were due to the lack of a standard national curriculum, specifically in the selection and sequencing of topics. High school courses were also weak in the treatment of debugging.

The studies lead Sleeman and his colleagues to start work on a series of diagnostic tests that teachers could use to quickly pinpoint student misconceptions.

SUMMARY

Computer programming was found to be a complex process, requiring a number of different teaching approaches from the instructor and skills and strategies from the students. While there was some mention of the problem definition and documentation steps, research was concentrated on algorithm design, code writing, and debugging.

PROBLEM DEFINITION

There was general consensus as to what the first step in program development was, but a definite lack of agreement on its scope. Linn described the first step as decomposing a complex problem into its components (1985, 14). Pintrich defined it as "a cognitive process that [included] generating some representation of the problem which [encompassed] the recognition of the type of programming problem presented and activating the relevant schemas or templates to begin solution" (453). Ingram claimed that the first thing a programmer had to do was to fully understand the process that related input in a program with the output from the program (216). Dalbey traced problems students had with planning to their lack of understanding of the problem specification. He stated that students needed additional instruction in interpreting "program specs" (1986a, 433).

Other than these few definitions and references, no research was found pertaining to what problem definition was composed of or how it was accomplished.

ALGORITHM DESIGN

The second step in the programming process was designing a solution. Dalbey stated that "design [was] the most important cognitive activity in programming" (1986b, 76), yet design skills were not being taught (Dalbey, 1986a, 431) and students were not engaged in planning activities (Dalbey, 1986b, 89).

Research in algorithm design centered around several distinct areas; skills required, approaches to instruction, and specific teaching tactics, including top-down design, structure diagrams, group settings and mental models.

Kurland stated that it was "not possible to identify a unique set of skills which [were] necessary for programming" (1984, 5). The cognitive demands depended partly on whether a linear or a modular style of programming was acceptable. With Logo and BASIC, it was possible for students to achieve desired output with brute force, linear programming, which demanded very low-level reasoning skills. Modular programming, on the other hand, required the students to learn to program with variables and procedures, to generate code that could be reusable, and to understand the control structure of the language. This required a much higher level of understanding of the language features (Kurland, 1984, 12).

Thus, the age of the programmer, the intended level of expertise, the characteristics of the language, the programming environment and the style of instruction all had an effect on what cognitive skills would be required of a programmer. Kurland did find that procedural reasoning was necessary for students to be

able to apply syntactic knowledge, evaluate the dynamic values of variables, translate word problems into symbolic statements, and design and comprehend an algorithm (1986, 450).

Van Merrienboer identified three prevalent instructional approaches to teaching introductory programming courses, two of which, the Expert approach and the Reading approach, appeared to emphasize the design process (253). In both of these strategies, students started out with fairly complex code which they were to read, understand, or interpret. They looked at the syntax rules only after having studied the program logic and flow of control. The third strategy, the Spiral approach, presented syntactic and semantic knowledge simultaneously in incremental steps, and therefore, design skills were not utilized in this method until quite late in the course (Van Merrienboer, 253-258).

Dalbey's Spider World programming environment could have been considered a variation of Van Merrienboer's Expert approach. Students were given complex, yet motivating, programs to work with before any BASIC syntax was introduced. Spider World was successful in that, after they started BASIC instruction, students learned syntax quickly and their attitude toward programming remained enthusiastic. It did not, however, appear to improve the students' design skills and after 12 weeks, students still could not design programs (Dalbey, 1986b, 89).

Teaching program design should have involved teaching problem solving, but as Ingram stated, "most instruction [concentrated] on the commands of a language and merely [demonstrated] by example the

techniques of problem solving" (216). Ingram designed an instructional model for teaching top-down programming. In a top-down design, a problem was subdivided into major subtasks. Each subtask was then subdivided into smaller tasks. This process was known as top-down design, or structured programming, and each successive subdivision was referred to as a stepwise refinement. Tasks at each stage of this process were called modules. This tactic Ingram referred to as an "heuristic" rather than an algorithm. "Heuristics," unlike algorithms, could not "be specified as precise rules, and they [could] not guarantee" a successful solution (217).

Experts had little trouble with this design process, drawing on their knowledge of the language syntax and their experience with describing a process for a machine solution (Dalbey, 1986a, 430). Novice programmers, on the other hand, experienced a great deal of difficulty with the design process. Students were more successful when they followed Ingram's four-step plan of 1) describing the problem, 2) dividing the problem into modules, 3) writing the algorithms for each module, and 4) translating each algorithm into code. Ingram reported from his studies that students found the step of breaking a problem into modules was the most perplexing. This was especially difficult if the practice problem was too small or trivial to be easily subdivided. Students also considered writing complete algorithms for short problems to be a waste of time (Ingram, 223).

Not everyone was a proponent of structured programming.

Cobern, for example, argued that one advantage of languages such as BASIC and Logo was the fact that they allowed a freeform style of coding for situations where a "planning in action" method was more appropriate than preplanning. This style would be analogous to the way an artist paints a picture, or a musician composes music (8).

Structured diagrams were suggested as a method of designing efficient code (Van Merrienboer, 265-66; Dalbey, 1986a, 431). These included any type of graphic representation of the flow of control of a program. There were many types of structured diagrams. The type Dalbey used during the Assessing the Cognitive Consequences of Computer Environments for Learning (ACCCEL) Project was a simple three-block scheme depicting the three basic constructs of action, loop and decision. The students had little trouble learning to apply the diagrams to simple problems, but had a difficult time with the more complex tasks. Dalbey suggested that students be given more complex programs early in a course, and that they be taught how to use structured diagrams as a tool for understanding flow of control processes (1986a, 433).

Another type of structure design was the flowchart. The use of flowcharts had declined since the late 1970s, but Scanlan contended that flowcharts may have had some value as an "aid in the comprehension of short, complex algorithms" (56). Good teaching techniques dictated that complex algorithms should be presented in as many ways as possible. Flowcharts not only presented a verbal solution of a problem, but a visual, graphic picture of the solution as well. In his study of college students, Scanlan found

that older students, those with higher G.P.A.'s and girls all preferred flowcharts over pseudocode (62).

Ross, on the other hand, presented flowchart instruction to high school students for three weeks and found that not only did they dislike flowcharting, but they, in fact, felt intimidated by it. They considered flowcharting to be a separate activity, not a part of the planning process, and often did the flowcharts after they did the program (Ross, 425). Ross suggested that flowcharting be introduced later in the curriculum, after the students had gained a fundamental grasp of the programming process (427).

Programming was a very social process in the professional setting, with programmers working in teams to complete systems of programs. In a school atmosphere, a teacher's concern with individual student accountability and evaluation often overshadowed this social aspect. MacGregor suggested that teachers borrow the structured walk-through technique from the professional programming setting as a way to emphasize both planning and student collaboration (7). It appeared that the students spent more time planning their programs when their work was to be critiqued by their fellow students. The end result was that the students then wrote more eloquent programs from their plans, spent less time debugging, and generally had a more positive attitude toward programming.

Webb also found that students were more successful with their own program generation if they participated in group discussions during the planning phase. It appeared that the group work

influenced the behavior of the partners rather than their knowledge, but the end result was achievement of better programs (Webb, 1986, 257).

Group collaboration was only one of several ideas taken from the professional setting. As part of the ACCCEL Project, Linn studied the activities of expert programmers. She found that as part of their design strategy, experts had a large repertoire of templates, which they could combine into procedures. Linn defined templates as "stereotypic sequences of computer instructions; they [could] be represented as code, pseudocode, or material language descriptions" (1987, 470). Dalbey defined templates as "chunks of code longer than a single line that [performed] reasonably complex functions....[They were] abstractions at a higher level than specific language features" (1986b, 77). Procedures were "actions such as seeking patterns, combining templates, etc." (Linn, 1987, 470). According to Linn, expert programmers combined templates in a procedure to solve a problem. Then their solution became a template, and their repertoire grew with their experience (1985, 16).

In her study, Linn found that experienced, successful teachers explicitly taught good, well-organized templates to their students (1985, 15). Dalbey found that having a repertoire of templates gave students a flexible design tool, encouraged more efficient code, reduced design time, and allowed students to write more complicated programs (1986b, 77). His tests showed that most students began to understand how to plan by using looping and

subroutine templates (1986b, 89). Van Merrienboer carried the idea of explicitly teaching templates one step further. He suggested using annotated, concrete, worked-out examples. He found it beneficial to include comments describing exactly what the examples were supposed to illustrate and how they could be used (268).

Along with explicitly teaching templates, experienced teachers also emphasized procedural skills. They modeled how to use reference materials to get information, required written plans before students were allowed at the machines, demonstrated to students how to design problem solutions, illustrated testing plans for exception cases, and discussed errors on assignments with the class (Linn, 1985, 26).

CODE WRITING

Once the programmer had the algorithm planned to solve the problem, it was time to write the code. Research provided studies of the skills required, approaches to instruction, instructional tactics, an analysis of student activities, group interaction, and evaluation methods.

Writing and editing the program code required not only a thorough understanding of the syntax and semantics of the programming language being used, but also an appreciation of the program's interrelationship and structure. Van Merrienboer referred to these two elements as declarative and procedural knowledge (260). Declarative instruction emphasized the acquisition of knowledge, and as such, preceded procedural instruction. Procedural instruction involved putting the knowledge to use (Van Merrienboer, 260). "The procedural knowledge base of an expert programmer [was] estimated to consist of ten to hundreds of thousands of highly task-specific productions, which all [had] to be acquired through the interpretative use of declarative knowledge" (Van Merrienboer, 270).

Magliaro's study seemed to agree with this idea of procedures building slowly on a knowledge base. Beginning students were able to memorize single lines of code, but had a difficult time with chunks, or templates. Intermediate and advanced students could recall chunks of code because they tried to make sense out of the chunks first, whereas, beginners had to rely on rote memorization skills (10-11).

There were different theories about whether syntax and semantics should be taught simultaneously or separately. Dyck's study showed that syntax and semantics did not need to be taught together. In fact, she was successful in teaching the semantics of coding first in English, then the syntax of BASIC (23).

Kagan found that there was a correlation between natural language skills and success in learning a programming language. Because of the similarities, she suggested instructors might borrow some ideas from the instructional methodology used to teach foreign languages. She also suggested that since writing and understanding were two different skills, teachers should not attempt to teach writing program code by having students study examples of templates (50). This was sharply in contrast with both Van Merrienboer's idea of supplying worked-out examples (253) and Pea's findings of negative transfer effects between the natural language and a computer language (1987, 7).

Various instructional practices were found to have a significant effect on student success. Linn's study of experienced teachers showed that they provided detailed feedback, engaged in explicit instruction, worked more with individuals or small groups, and spent little time in group lectures (1985, 26).

Feedback had a direct effect on student performance. According to Linn, feedback consisted of teachers writing comments on completed assignments; returning assignments promptly; explaining to small groups or individuals how to improve programs; providing a solution to the assignments; and describing different

ways the assignment could have been solved. Students were more successful with more detailed feedback (1987, 486-87).

Linn found that computer access time had a direct positive correlation with programming success for all students at all levels. She cautioned, however, that increased time-on-task occurred simultaneously with increased individual attention from the teacher, and her study did not distinguish which activity had the greater effect (1987, 476). Ross found that low ability students benefitted from unlimited access time, while medium and high ability students preformed better in a restricted access environment (426). Other studies found unlimited access time prohibitive. Kurland warned that high access to computers and the immediate interactive nature of the programming environment may have tended to foster trial and error programming rather than well-planned, efficient code (1984, 31).

A great amount of students' class time was spent at the computer. Pintrich found that the majority of students' computer time was spent listing, running and debugging their programs. They spent very little time entering code, and almost no time planning (458). Kurland discovered that most students, after two years experience, still produced simple, linear code using brute force programming, and showing no evidence of design or planning (1986, 452). Both Pintrich and Kurland found that students borrowed a lot of code, either from other students, or from textbooks (Pintrich, 456; Kurland, 1986, 453). While sharing code was an acceptable practice among professionals, when students borrowed code and used

it without trying to understand it, or when they never made contributions of original code to the group, problems occurred.

Webb found no difference in the programming outcome among students working individually and those working in pairs. She did, however, find considerable variation between the setting, (group or individual), the type of programming outcome, (syntax knowledge, code interpretation, or code generation), and student's cognitive characteristics, (spatial, mathematics, verbal inference, nonverbal reasoning and age) (1985, 189). In a second study, Webb reported that students did well working in groups on planning and debugging strategies if they discussed plans with each other and not with the instructor. Webb suggested that the teacher give students in a group very explicit, structured guidelines, and then intervene as little as possible to force the students to verbalize their strategies with each other (1986, 252).

One final area that could be considered part of the coding process was teacher evaluation of student accomplishments. Several ideas were presented on ways to test student's understanding. By far the most commonly used testing vehicle was some type of comprehensive objective exam. The types of questions suggested were varied to test all levels of understanding of either syntax, flow of control features or functions of the language. Students were asked to recognize and interpret valid statements (Dalbey, 1986b, 76). They were also expected to be able to identify the purpose and predict the output of already coded segments, varying from simple templates to complex programs (Foreman, 424).

Reformulation of programs involved making minor changes to slightly modify either the output or function of presented code (Dalbey, 1986b, 76). Ross tested error recognition by presenting short sections of code with either syntax or logic errors. The students were to find the error, specify the cause and make the correction (424). Code generation questions could test for higher levels of the cognitive domain, depending on the complexity of the problem to be coded.

As an alternative to comprehensive exams, Cook investigated using the "Cloze" procedure, a type of fill-in-the-blank method, where students were required to supply missing parts of a program (199). This was the only study found where the testing implement itself was actually investigated.

DEBUGGING

Before a program could be considered complete, all the syntax and logic errors had to be removed through a debugging process. Research revealed different approaches to the debugging process, new programming environments available, and studies of student misconceptions that lead to errors.

"Debugging the program [involved] the various strategies used to modify the program to have it operate properly" (Pintrich, 453). Studies showed that students did not have techniques or rules for systematically analyzing buggy programs for developing corrections (Kurland, 1986, 453). There were several different approaches to take to debugging. Klahr proposed a very specific, five-step model which his students successfully used to reduce their debugging time. The students were taught to 1) compare planned and actual output, 2) identify the type of bug, 3) isolate the suspected buggy code, 4) locate the bug, and 5) correct the bug (366-69).

Linn found in her study that experienced programmers could actually identify program bugs in a matter of seconds just from looking at the output. For example, if a numerical value was grossly inaccurate, the most likely problem would be an uninitialized variable; if the output was close to being correct, a loop process could have been off by one (1987, 472-73). Linn verified that experts had this ability, but no investigation was done on how they did it. Kurland called this skill decentration, being able to tell the difference between what the program should do and what the computer actually did (1984, 8).

Linn likened debugging instruction to teaching students how to use reference materials and library services efficiently (1985, 17). She emphasized explicitly teaching testing skills. Students were taught to test typical cases, boundary cases, interaction between parts, bad input, lack of input, input in wrong order, loops for off-by-one, end elements in array handling and sort routines with only two elements or identical elements (Linn, 1987, 472).

Pintrich identified four different behaviors involved in debugging. Typing mistakes and simple syntax errors were corrected by making low level changes in the code. A trial and error approach to debugging involved changing one value and re-running the program to see what happened. Decomposing code was going through the program very carefully and trying to isolate the errors. This included checking subroutines, tracing program flow, and obtaining and examining printouts of variable values. In reformulating code, the most complex behavior, the student made major revisions, at least five lines, to the code (453-464).

Pea suggested that teachers needed to adapt research methods to help students find bugs in their programs. He used programs specifically designed with problems and had students do hand-simulations to predict the output. He also had students think aloud as they planned or wrote programs. He then used the students answers as an expository tool to teach other students what errors to avoid (1987, 24-25).

New programming environments may have changed the cognitive

skills needed for debugging. The new editors that were reviewed had good trace routines, windows displaying both the code and the corresponding output simultaneously on the screen, and step-by-step execution that displayed the changing values of variables as the program executed. Linn described the features of MacPascal, a new Pascal editor (1987, 475). Kurland used Interlisp-D as a Lisp processor (1984, 5). These new programming environments helped the students to visualize their static program code as it turned into a dynamic computational structure at run-time.

Newby described a new system, ALICE, that not only had all the new debugging features, but also had a very sophisticated Pascal entry feature. The editor displayed place-holding templates on the screen as you typed in key words, then checked for syntax errors and type compatibility as the text was entered (53-54).

Debugging was a time consuming process. Pintrich found that Pascal students spent as much as 47% of their lab time on debugging-related activities (458). If teachers could anticipate student errors and teach to them, debugging time could be cut down significantly (Sleeman, 3). Considerable research has been done lately in identifying the source of student errors.

Pea stated that students were "actively building a knowledge system of concepts and procedural skills" (1987, 5). They used this knowledge base to try to make sense of the new problem solving activities in programming. Pea contended that most student errors stemmed from their misapplication of this base knowledge rather than from syntactic errors (1987,6). Pea, Sleeman and Samurcay

have all identified numerous misconceptions that students have of the nature of the computer and the syntax and semantics of the languages.

Samurcay studied the students use of their existing mathematical models in understanding programming concepts. In many instances, the model was acceptable, but there were differences that the students needed to be explicitly made aware of. In math, for example, the equal symbol meant equality, but as part of a programming assignment statement, it displayed an asymmetric relationship (Samurcay, 38). A variable had a unique value in math that remained static throughout the problem, while, in programming, a variable often had a dynamic changing value. In the accumulation process, n = n + 1, for example, the same variable designated both a previous and a present value in the same statement (Samurcay, 38). The dynamic status of a variable caused problems for students in looping constructs. They had difficulty planning the transformation of a variable from its initial value to its final value (Samurcay, 39). Students often failed to initialize variables in Pascal and other languages because of the same misconception (Samurcay, 42).

Pea studied the negative effects of the transfer between natural languages and programming languages. He found that while students didn't really believe that a computer could function as a human mind, when they had a problem with a code segment, they reverted back to human language rules because they didn't have another frame of reference (1987, 13). The "parallelism bug", the

"intentionality bug" and the "egocentrism bug" were three misconceptions that Pea attributed to the student's assignment of natural language characteristics to the computer (1986, 27-30).

The entire process of computer programming was prone to student misconceptions, and Pea identified and labeled a number of them. "Knowledge unavailability bugs" stemmed from a lack of syntactic or semantic knowledge, an inadequate mental model of how a computer mechanically interpreted code, a lack of understanding of flow of control processes, or a lack of knowledge of program structure (1987, 22-23).

"Knowledge inaccessibility bugs" were a result of weak or faulty analogical reasoning skills (Pea, 1987, 24). Students had a knowledge base of plans, or templates, but didn't know when to use them, or how to merge several plans together into a coherent program.

Kurland's work confirmed the findings of Pea and Samurcay. He found that even in their second year, students could not systematically keep track of values of variables, especially if the variables were defined in terms of other variables. They had problems evaluating conditional tests that controlled the iterative constructs. They could not predict when a loop would end, much less control the process in their own code. Students also had difficulty understanding how the order of execution of the lines in a program was determined by particular flow of control commands and the current value of variables (Kurland, 1986, 452-53).

Pea suggested several concepts teachers had to stress to help

students correct their misunderstandings. To encourage analogical reasoning, teachers should have presented new concepts using a variety of examples so that students were less likely to associate the procedure with only one type of problem (Pea, 1987, 24). Teachers needed to emphasize the strict flow of control the computer adhered to. Only one line was executed at a time. Only current values of variables were applicable at that instant. The computer neither remembered what had occurred nor anticipated what was to come (Pea, 1987, 10). Finally, teachers needed to stress the mechanical aspect of controlling a machine through language commands. In programming, the sole responsibility for clear, concise, explicit communication rested with the programmer. Human communicators interpreted remarks, filled in gaps, and repaired ambiguities based on a common social background. The computer interpreted the instructions using a mechanical process outlined by rigid language rules (Pea, 1987, 8).

Sleeman reported three successful methods teachers could use to remediate many of the students' misconceptions. First, students benefitted from explicit training about syntax and semantics of specific constructions (3). To help teachers select areas that needed direct instruction, Sleeman and his colleagues started work on a series of diagnostic tests to identify student errors (5). Second, teachers needed to require learners to make predictions about the outcome of short programs. The students should have learned to read programs before they attempted to write them (Sleeman, 3). This corresponded with the Reading approach to

instruction that Van Merrienboer found the most effective (257). Third, students should have been provided with interactive feedback from the computer (Sleeman, 3).

DOCUMENTATION

No research could be found on the subject of documenting computer programs. Van Merrienboer required explicit documentation as part of his annotated worked-out examples (253). Pea hypothesized that careful program documentation would be a result of "metacognitive activities involving reflection on the state of one's knowledge and skills in programming" and debugging (1987, 14).

Of the five studies that listed steps in programming (Kurland, 1984, 2; Goktepe, 168; Webb, 1985, 186; Dalbey, 1985, 254; Moursund, 4), only Maursund listed documentation as a separate step in the process.

CONCLUSION

Research related to the teaching of computer programming has centered around designing algorithms, writing code and debugging programs. Some of the theories presented included the following:

1. Modular programming was more effective than linear programming because it required high-level reasoning features, such as variables, procedures, reusable code and control structures.
2. Two instructional approaches, the Expert and the Reading approaches, emphasized design skills more than the Spiral approach. The Reading approach appeared to be the most compatible with effective instructional tactics.
3. Novice programmers needed a specific model of the top-down programming process.
4. Structured diagrams were successful in some classes, giving the students an optional method to use in following the flow of control of complex algorithms.
5. Flowcharts were a useful tool to some students, as they depicted a visual, graphic picture of a solution. They seemed confusing, however, if presented too early in the course, before problems were complex enough to require planning.
6. Successful programmers had a repertoire of templates to serve as design tools, enabling them to produce more efficient code in less time.
7. Experienced teachers emphasized and modeled procedural skills such as using reference materials, writing solution plans

before coding, and using test patterns.

8. Students needed a good syntactic knowledge base before they could generate code, but they could learn the semantics of a language independent of the syntax.
9. Students were more successful with immediate, detailed feedback from the teachers, such as assignments quickly returned, suggestions on improving their code, and examples of different solutions to the problem.
10. Unlimited computer access did not always produce more successful student behavior. More time-on-task seemed to be related to planned activities and/or individual teacher attention.
11. In a professional setting, programming had a social aspect. Programmers worked together and borrowed code from other sources. Group work appeared effective in the classroom if the groups were given clear, specific guidelines as to what was to be accomplished by the group and what was expected of each individual.
12. Students were more successful in program generation if they participated in small group planning discussions.
13. Numerous types of test questions could be used to evaluate various levels of achievement. Program generation was considered the most difficult type of question. The "cloze" procedure provided easy-to-construct, effective tests.
14. Students were not able to systematically analyze or correct buggy programs. If left on their own, they engaged in a trial

and error method of debugging.

15. Students did not test a program further for logic errors once the program ran successfully.
16. Existing mathematical models caused misconceptions about programming structures. These could be removed if teachers recognized the problem areas and explicitly pointed out the differences between mathematical and computer processes.
17. Natural language was also a source of student misconceptions. When faced with a problem and no solution model, students reverted to human language skills, unconsciously attributing the computer with human communication skills.

Considering the scope and complexity of the field, the research has been scant, with some areas not yet even being considered. Many of the studies sampled only a few students or teachers, and reported inconclusive results.

Kurland sums up the situation computer programming teachers find themselves in as follows:

> We know far too little about what to expect students will learn in a reasonable period of time, how they learn, what conceptual difficulties they encounter, what forms of cognitive support they may require to guide them over these difficulties, or whether individual differences in learning styles are reflected in programming and need to be guided differently in instruction. (1986, 430)

RECOMMENDATIONS

What then constitutes effective teaching of computer programming? To answer this, teachers will have to engage in investigative practices themselves. Even if many of the current studies are inconclusive, they contain a wealth of ideas.

1. Students should be encouraged to engage in modular programming.
2. Teachers could emphasize design skills more by implementing a instructional approach that requires students to read and interpret large, complex programs right from the start.
3. Students could be required to use a specific model of the top-down programming process.
4. Teachers could use structured diagrams in the presentation of complex algorithms.
5. Flowcharts could be used as an alternate, visual method of planning program solutions.
6. Teachers should explicitly teach templates, starting with short, well-documented examples of simple solution patterns, and helping students learn to combine these into their own repertoire of design tools.
7. Procedural skills should be emphasized. Reference material should be available and teachers should model how to use it. Written plans could be required, either in pseudocode, flowcharts, or structure diagrams, before students are allowed at the machines to code.

8. The semantics of a language can be taught first, illustrating the program design, the flow of control and the machine's interpretive process. Then, when students can understand why and how a procedure works, they can be introduced to the specific syntax constructs.
9. Teachers need to provide immediate, detailed feedback on student assignments.
10. Students don't do well if just turned loose in a lab. They should either have a plan to follow; or the teacher should be prepared to provide a lot of individual or small group attention.
11. Teachers should occasionally allow group planning of some projects.
12. Teachers need to provide students with a clear set of guidelines as to the individual effort that will be required of them. Teachers must determine what is original code and what is copied, whether students would be allowed to help each other in lab, and when group planning or debugging would be appropriate.
13. When constructing tests, teachers should keep in mind that program generation questions tend to test a level of student cognitive development not often reached by novice programmers.
14. Students should be given a model to use for debugging programs. The decentering skill, being able to distinguish between what the program should do and what the computer actually did, needs to be developed with explicit examples.

15. There are very specific test cases that students should be required to try before considering their program complete.
16. Teachers should explicitly point out the difference between mathematical equality and program assignments, and between static variables in math and dynamic variables in programming. Students have difficulty visualizing and planning the transformation of a variable, especially in a looping construct.
17. To clear up misconceptions and encourage debugging skills, teachers need to require students to read programs and predict the output, closely track the changing status of variables during execution, and strictly follow the flow of control to which the computer adheres.

Teachers should feel free to take different approaches and try new methods. Until the teaching methodology of computer programming is better defined, all classes could and should become research centers.

WORKS CITED

Cobern, William W. "Programming Language Choice: A Positive albeit Ambiguous Case for BASIC Programming in Secondary Science Teaching," 1986.

Cook, Curtis & Others. "A Preliminary Investigation of the Use of the Cloze Procedure as a Measure of Program Understanding," Information Processing and Management, 1984, 20 (1-2) 199-208.

Dalbey, John and Linn, Marcia C. "The Demands and Requirements of Computer Programming: A Literature Review," Journal of Educational Computing Research, 1985, 1 (3) 253-74.

Dalbey, John & Others. "Making Programming Instruction Cognitively Demanding: An Intervention Study," Journal of Research in Science Teaching, May, 1986a, 23 (5) 427-36.

Dalbey, John & Marcia C. Linn. "Cognitive Consequences of Programming: Augmentations to BASIC Instruction," Journal of Educational Computing Research, 1986b, 2 (1) 75-93.

----- Dictionary of Instructional Technology, compiled by Ellington, Henry and Harris, Duncan. New York, Nichols Publishing Company, 1986.

Dyck, Jennifer L. and Mayer, Richard E. "Teaching for Transfer of Computer Program Comprehension Skill," Journal of Educational Psychology, Mar. 1989, 81 (1), 16-24.

Foreman, Kim Hyun Deok. "Cognitive Style, Cognitive Ability, and the Acquisition of Initial Computer Programming Competence," 1988.

Good, Carter Victor, ed. Dictionary of Education, 3rd Edition, New York, McGraw-Hill, 1973.

Goktepe, Mesut & Others. "Design and Implementation of a Tool for Teaching Programming," Computers and Education, 1989, 13 (2) 167-78.

Ingram, Albert L. "Instructional Design for Heuristic-Based Problem Solving," Educational Communication and Technology Journal, Win. 1988, 36 (4), 211-30.

Kagan, Dona K. "Learning How to Program or Use Computers: A Review of Six Applied Studies," Educational Technology, Mar. 1988, 28 (3) 49-51.

Klahr, David and Carver, Sharon McCoy. "Cognitive Objectives in a Logo Debugging Curriculum: Instruction, Learning and Transfer," Cognitive Psychology, Jul. 1988, 20 (3) 362-404.

Kurland, D. Midian, Editor. "Developmental Studies of Computer Programming Skills. A Symposium: Annual Meeting of the American Educational Research Association," 1984.

Kurland, D. Midian & Others. "A Study of the Development of Programming Ability and Thinking Skills in High School Students," 1986, 2 (4) 429-58.

Linn, Marcia C. "The Cognitive Consequences of Programming Instruction in Classrooms," Educational Researcher, May, 1985, 14 (5) 14-16, 25-29.

Linn, Marcia C. and others. "Ideal and Actual Outcomes from Precollege Pascal Instruction," Journal of Research in Science Teaching, May, 1987, 24 (5) 467-90.

Linn, Marcia C. "Autonomous Classroom Computer Environments for Learning. Progress Report and Annotated Bibliography," May 1988, 1-13.

MacGregor, S. Kim. "The Structured Walk-Through," Computing Teacher, Jun. 1988, 15 (9) 7-10.

Magliaro, Susan & John K. Burton. "Adolescents' Chunking of Computer Programs," 1986.

Moursund, David. "Problem Solving: A Computer Educator's Perspective," The Computing Teacher, Feb. 1985, 3-5.

Newby, Michael. "Teaching Introductory Programming Using a Language-Intelligent Programming Environment," in "Using Computers Intelligently in Tertiary Education. A Collection of Papers Presented to the Australian Society for Computers in Learning," Sydney, Australia, 1987, 52-57.

Pea, Roy D. "Language Independent Conceptual 'Bugs' in Novice Programming," Journal of Educational Computing Research, 1986, 2 (1) 25-36.

Pea, Roy D. "The Buggy Path to the Development of Programming Expertise," Focus on Learning Problems in Mathematics, Win. 1987, 9 (1) 5-30.

Pintrich, Paul R. & Others. "Students' Programming Behavior in a Pascal Course," Journal of Research in Science Teaching, May, 1987, 24 (5) 451-66.

Ross, Steven M. & Deborah McCormick. "Computer Access and Flowcharting as Variables in Learning Computer Programming," 1989.

Samurcay, Renan. "Learning Programming: An Analysis of Looping Strategies Used by Beginning Students," For the Learning of Mathematics-An International Journal of Mathematics Education, Feb. 1985, 5 (1) 37-43.

Scanlan, David & Leisa Clark. "An Empirical Investigation of Flowchart Preference," Journal of Computers in Mathematics and Science Teaching, Win. 1988-89, 2, 56-64.

Sleeman, D. & Brian Gong. "From Clinical Interviews to Policy Recommendations: A Case Study in High School Computer Programming. Study of Stanford and the Schools Technology Panel," 1985.

Van Merrienboer, Jeroen J. G. & Hein P. M. Krammer. "Instructional Strategies and Tactics for the Design of Introductory Computer Programming Courses in High School," Instructional Science, 1987, 16 (3) 251-85.

Webb, Noreen W. "Cognitive Requirements of Learning Computer Programming in Group and Individual Settings," AEDS Journal, Spr. 1985, 18 (3) 183-94.

Webb, Noreen M. & Others. "Problem Solving Strategies and Group Processes in Small Groups Learning Computer Programming," American Educational Research Journal, Sum. 1986, 23 (2) 243-61.